PYTHON 101 & MORE...

AIOTLAB, September 2023

Table of Contents

1. Lambda Functions
2. Arrays
3. List
4. Sets
5. Tuples
6. Dictionaries
7. Short Summary

Table of Contents

1 Lambda Functions

2 Arrays

3 List

4 Sets

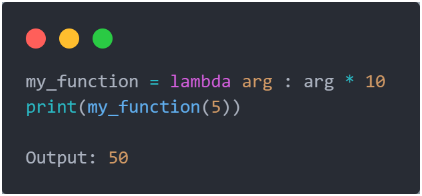
5 Tuples

6 Dictionaries

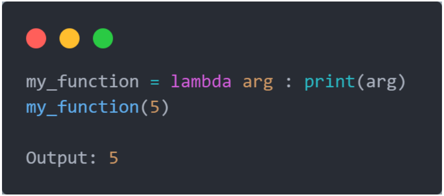
7 Short Summary

Lambda Functions

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.



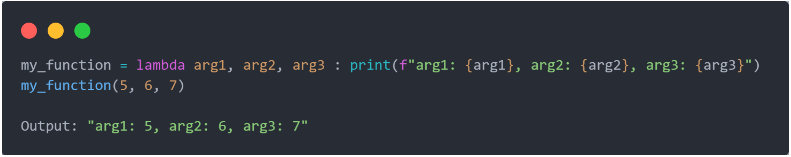
```
my_function = lambda arg : arg * 10
print(my_function(5))

Output: 50
```



```
my_function = lambda arg : print(arg)
my_function(5)

Output: 5
```

Lambda Functions

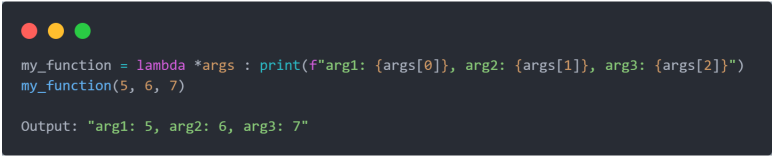
- Example of Lambda Functions can take any number of arguments



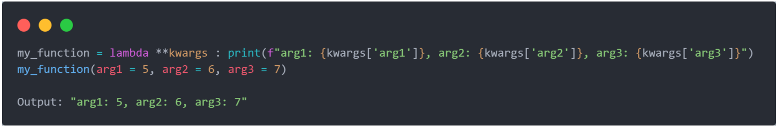
```
my_function = lambda arg1, arg2, arg3 : print(f"arg1: {arg1}, arg2: {arg2}, arg3: {arg3}")
my_function(5, 6, 7)

Output: "arg1: 5, arg2: 6, arg3: 7"
```

Lambda Functions

- Lambda function can use kwargs and args

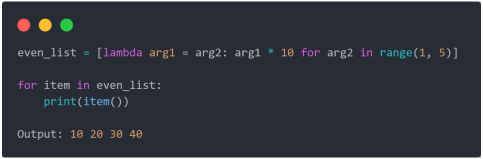


```
my_function = lambda *args : print(f"arg1: {args[0]}, arg2: {args[1]}, arg3: {args[2]}")
my_function(5, 6, 7)

Output: "arg1: 5, arg2: 6, arg3: 7"
```



```
my_function = lambda **kwargs : print(f"arg1: {kwargs['arg1']}, arg2: {kwargs['arg2']}, arg3: {kwargs['arg3']}")
my_function(arg1 = 5, arg2 = 6, arg3 = 7)

Output: "arg1: 5, arg2: 6, arg3: 7"
```

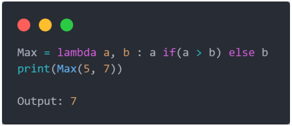
- Python Lambda function with list comprehentsion



```
even_list = [lambda arg1 = arg2: arg1 * 10 for arg2 in range(1, 5)]

for item in even_list:
    print(item())

Output: 10 20 30 40
```

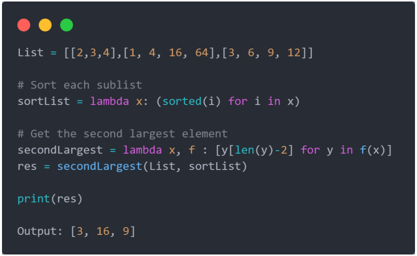
Lambda Functions

- Lambda function with if-else



```
Max = lambda a, b : a if(a > b) else b
print(Max(5, 7))

Output: 7
```

- Lambda with multiple statements: Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function.



```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]

# Sort each sublist
sortList = lambda x: (sorted(i) for i in x)

# Get the second largest element
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]
res = secondLargest(List, sortList)

print(res)

Output: [3, 16, 9]
```

Table of Contents

1. Lambda Functions
2. Arrays
3. List
4. Sets
5. Tuples
6. Dictionaries
7. Short Summary

Arrays

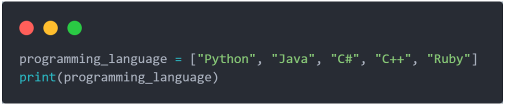
- Python does not have built-in support for Arrays, but Python Lists can be used instead.
- To work with arrays in Python you will have to import a library, like the NumPy library.

Table of Contents

1. Lambda Functions
2. Arrays
3. List
4. Sets
5. Tuples
6. Dictionaries
7. Short Summary

Lists

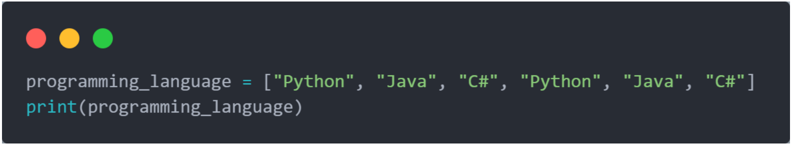
- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- Lists are created using square brackets:



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language)
```

- List items are indexed, the first item has index [0], the second item has index [1] etc.
- List items are ordered, changeable, and allow duplicate values.
  - When we say that lists are ordered, it means that the items have a defined order, and that order will not change.If you add new items to a list, the new items will be placed at the end of the list.
  - The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
  - Since lists are indexed, lists can have items with the same value

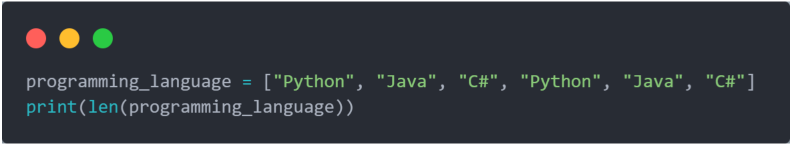
Lists



```python
programming_language = ["Python", "Java", "C#", "Python", "Java", "C#"]
print(programming_language)
```

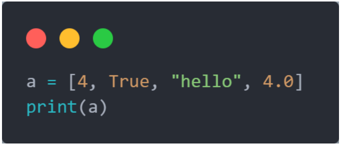
Lists

- To determine how many items a list has, use the **len()** function



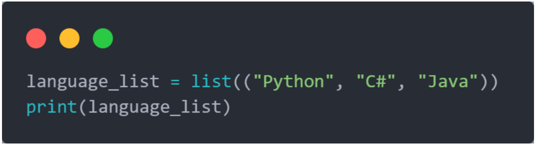
```
programming_language = ["Python", "Java", "C#", "Python", "Java", "C#"]
print(len(programming_language))
```

- List items can be of any data type
- A list can contain different data types



```
a = [4, True, "hello", 4.0]
print(a)
```

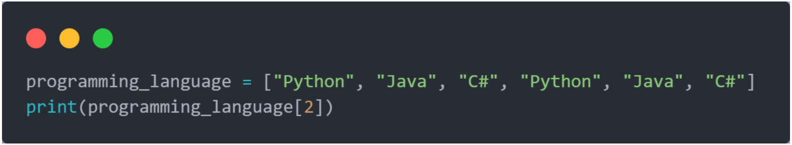
- It is also possible to use the **list()** constructor when creating a new list



```
language_list = list(("Python", "C#", "Java"))
print(language_list)
```

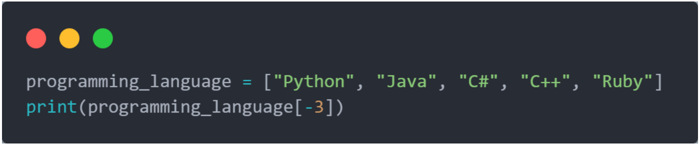
Lists

- List items are indexed and you can access them by referring to the index number



```
programming_language = ["Python", "Java", "C#", "Python", "Java", "C#"]
print(programming_language[2])
```

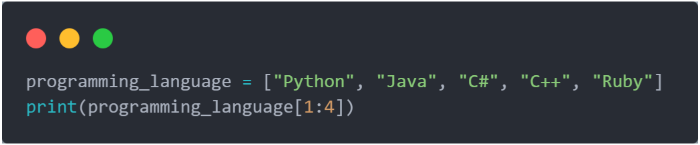
- Negative indexing means start from the end
- -1 refers to the last item, -2 refers to the second last item etc.



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[-3])
```

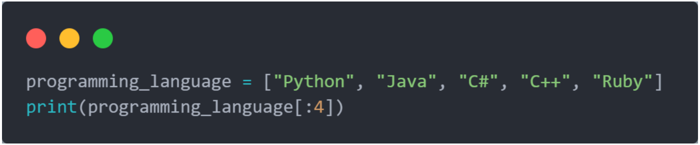
Lists

- You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[1:4])
```

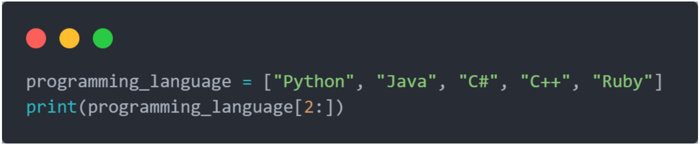
- By leaving out the start value, the range will start at the first item



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[:4])
```

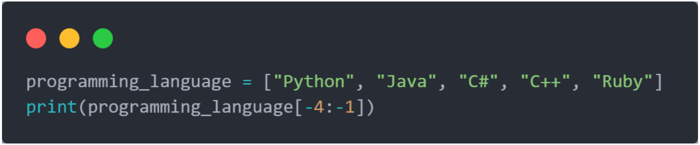
Lists

- By leaving out the end value, the range will go on to the end of the list



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[2:])
```

- Specify negative indexes if you want to start the search from the end of the list

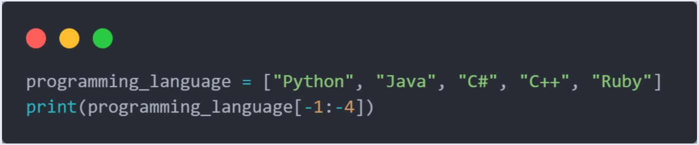


```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[-4:-1])
```

Lists

Question

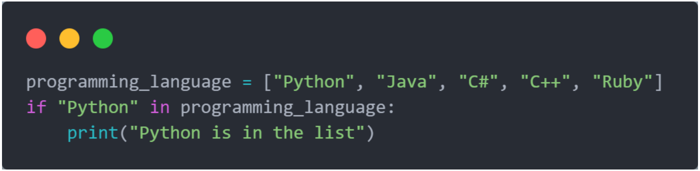
What is the result of the below code block?



```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
print(programming_language[-1:-4])
```

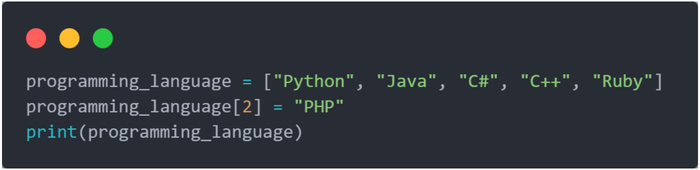
Lists

- To determine if a specified item is present in a list use the keyword: **in**



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
if "Python" in programming_language:
    print("Python is in the list")
```

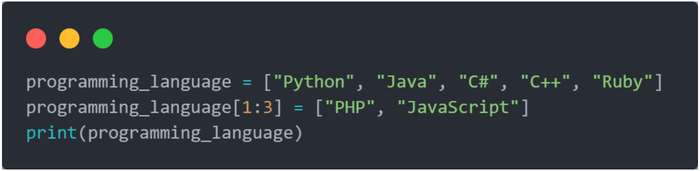
- To change the value of a specific item, refer to the index number



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[2] = "PHP"
print(programming_language)
```

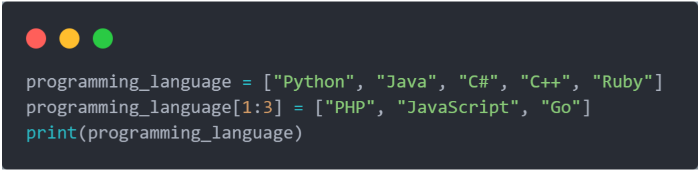
Lists

- To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[1:3] = ["PHP", "JavaScript"]
print(programming_language)
```

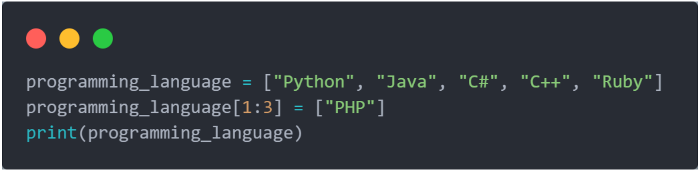
- If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[1:3] = ["PHP", "JavaScript", "Go"]
print(programming_language)
```

Lists

- If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly

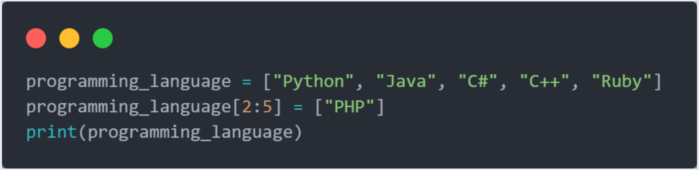


```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[1:3] = ["PHP"]
print(programming_language)
```

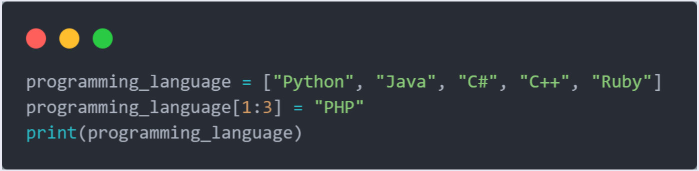
Lists

Question

What is the result of the below code block?



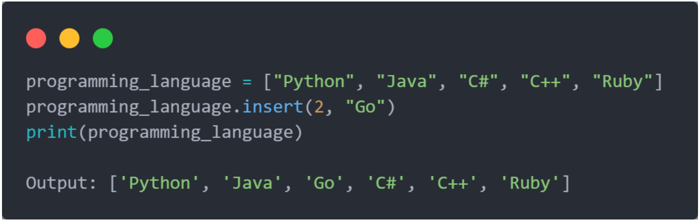
```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[2:5] = ["PHP"]
print(programming_language)
```



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language[1:3] = "PHP"
print(programming_language)
```

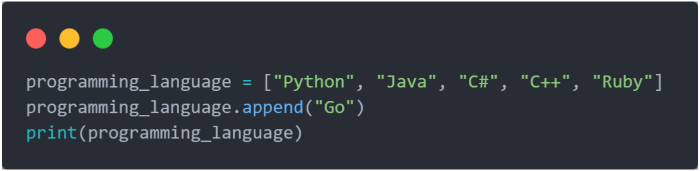
Lists

- To insert a new list item, without replacing any of the existing values, we can use the **insert()** method. The **insert()** method inserts an item at the specified index



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.insert(2, "Go")
print(programming_language)

Output: ['Python', 'Java', 'Go', 'C#', 'C++', 'Ruby']
```

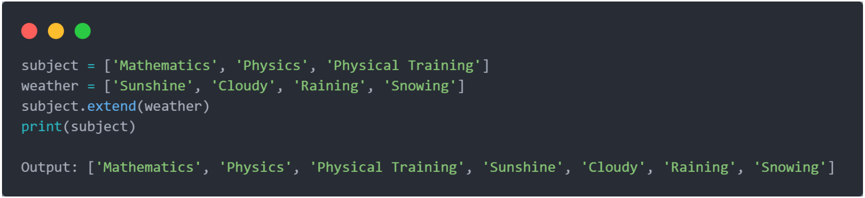
- To add an item to the end of the list, use the **append()** method



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.append("Go")
print(programming_language)
```

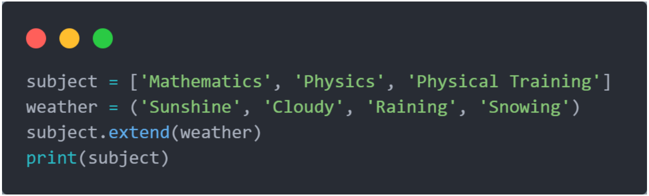
Lists

- To append elements from another list to the current list, use the **extend()** method



```
subject = ['Mathematics', 'Physics', 'Physical Training']
weather = ['Sunshine', 'Cloudy', 'Raining', 'Snowing']
subject.extend(weather)
print(subject)

Output: ['Mathematics', 'Physics', 'Physical Training', 'Sunshine', 'Cloudy', 'Raining', 'Snowing']
```

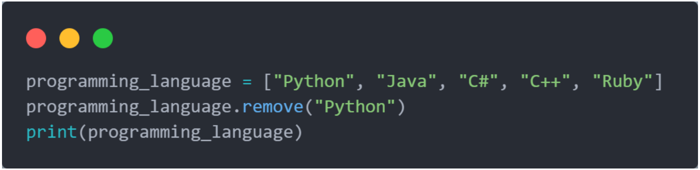
- The **extend()** method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.)



```
subject = ['Mathematics', 'Physics', 'Physical Training']
weather = ('Sunshine', 'Cloudy', 'Raining', 'Snowing')
subject.extend(weather)
print(subject)
```

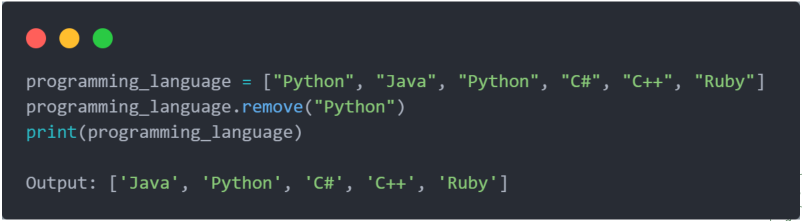
Lists

- The **remove()** method removes the specified item



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.remove("Python")
print(programming_language)
```

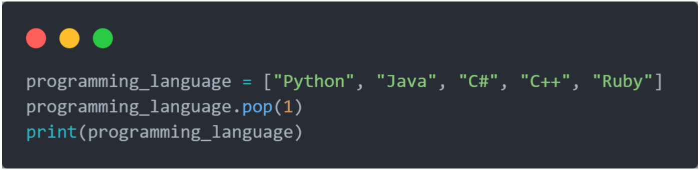
- If there are more than one item with the specified value, the **remove()** method removes the first occurance



```
programming_language = ["Python", "Java", "Python", "C#", "C++", "Ruby"]
programming_language.remove("Python")
print(programming_language)

Output: ['Java', 'Python', 'C#', 'C++', 'Ruby']
```

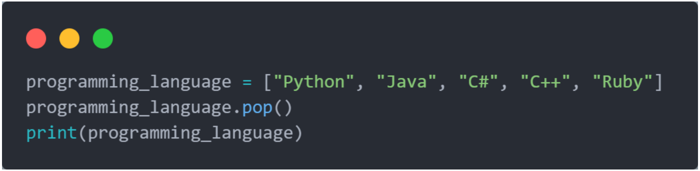
Lists

- The **pop()** method removes the specified index



```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.pop(1)
print(programming_language)
```

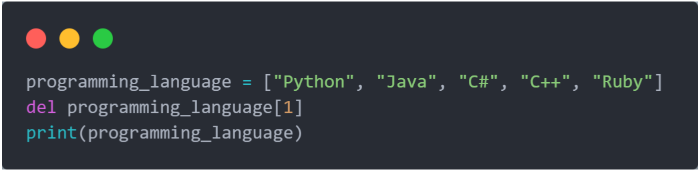
- If you do not specify the index, the **pop()** method removes the last item



```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.pop()
print(programming_language)
```

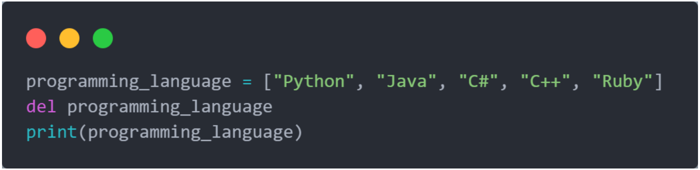
Lists

- The **del** keyword also removes the specified index



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
del programming_language[1]
print(programming_language)
```

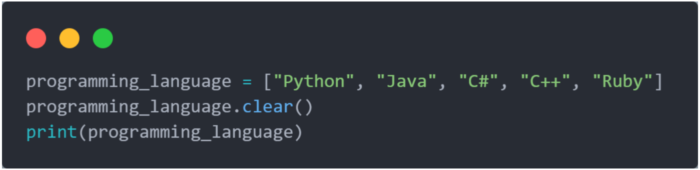
- The del keyword can also delete the list completely



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
del programming_language
print(programming_language)
```

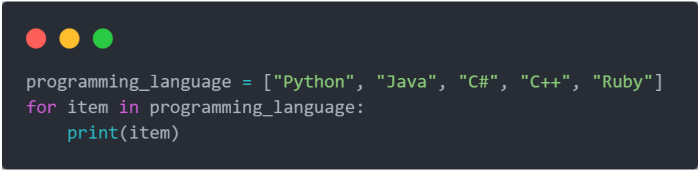
Lists

- The **clear()** method empties the list. The list still remains, but it has no content.



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
programming_language.clear()
print(programming_language)
```

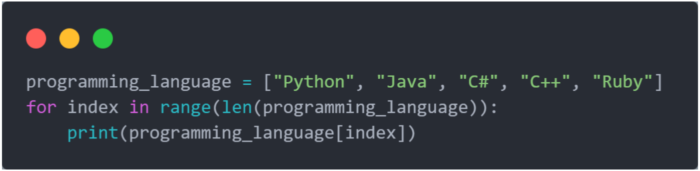
- You can loop through the list items by using a **for** loop



```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
for item in programming_language:
    print(item)
```

Lists

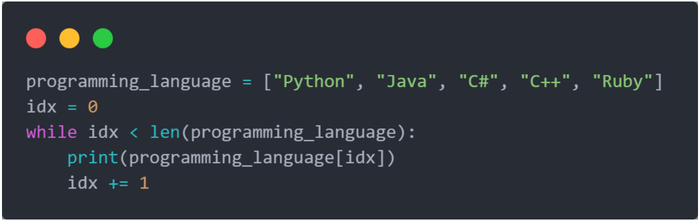
- You can also loop through the list items by referring to their index number. Use the **range()** and **len()** functions to create a suitable iterable.



```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
for index in range(len(programming_language)):
    print(programming_language[index])
```

Lists

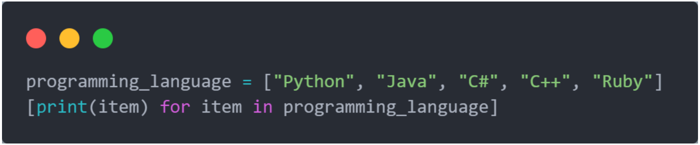
- You can loop through the list items by using a **while** loop. Use the **len()** function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.



```python
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
idx = 0
while idx < len(programming_language):
    print(programming_language[idx])
    idx += 1
```

Lists

- List Comprehension
  - List Comprehension offers the shortest syntax for looping through lists (short hand **for** loop)

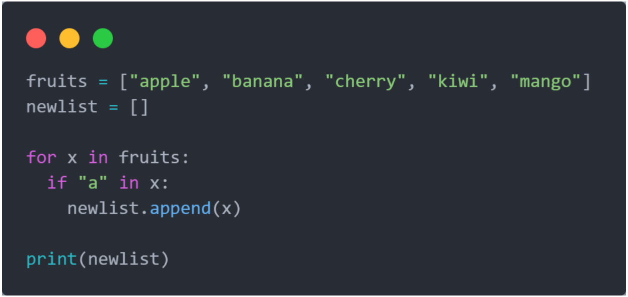


```
programming_language = ["Python", "Java", "C#", "C++", "Ruby"]
[print(item) for item in programming_language]
```

  - Another example of list comprehension: Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Lists

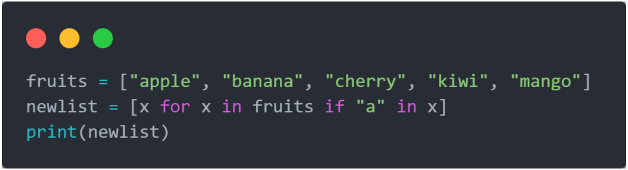
- Without list comprehension you will have to write a for statement with a conditional test inside



```python
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Lists

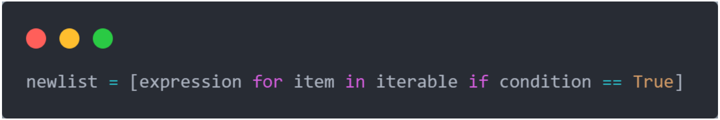
- With list comprehension you can do all that with only one line of code



```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

Lists

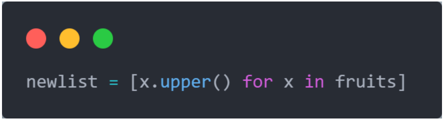
- Syntax for list comprehension



```python
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

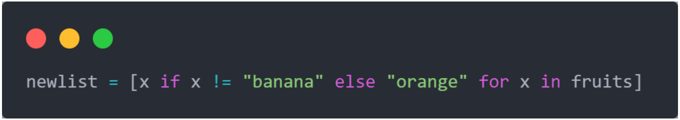
- The condition is like a filter that only accepts the items that valuate to **True**
- The iterable can be any iterable object, like a list, tuple, set etc.
- The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.



```python
newlist = [x.upper() for x in fruits]
```

Lists

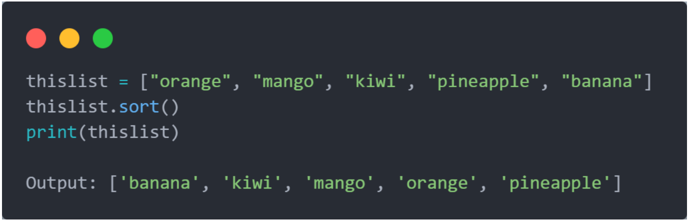
- The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome



```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

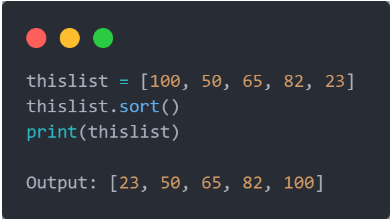
Lists

- List objects have a **sort()** method that will sort the list alphanumerically, ascending, by default



```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)

Output: ['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

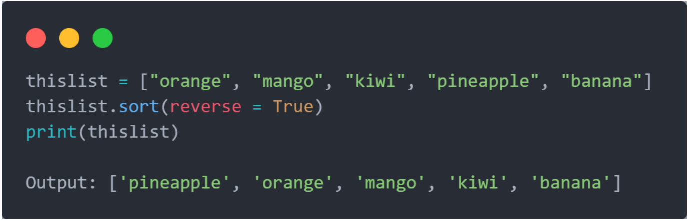
- Sort the list numerically:



```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)

Output: [23, 50, 65, 82, 100]
```

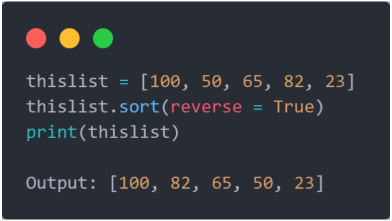
Lists

- To sort descending, use the keyword argument **reverse = True**



```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)

Output: ['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

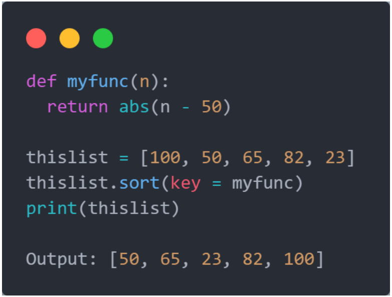
- Sort the list descending



```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)

Output: [100, 82, 65, 50, 23]
```

Lists

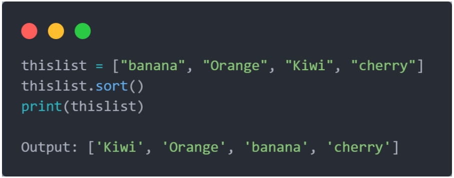
- Customize Sort Function
  - You can also customize your own function by using the keyword argument **key = function**
  - The function will return a number that will be used to sort the list (the lowest number first)
  - Sort the list based on how close the number is to 50



```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)

Output: [50, 65, 23, 82, 100]
```

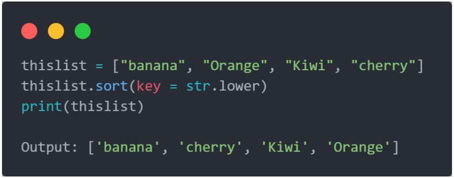
Lists

- Case Insensitive Sort
  - By default the **sort()** method is case sensitive, resulting in all capital letters being sorted before lower case letters



```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)

Output: ['Kiwi', 'Orange', 'banana', 'cherry']
```

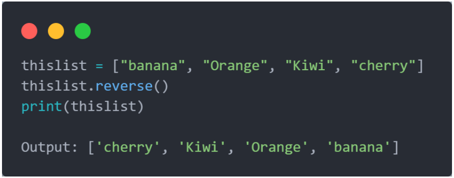
  - Luckily we can use built-in functions as key functions when sorting a list.
  - So if you want a case-insensitive sort function, use str.lower as a key function



```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)

Output: ['banana', 'cherry', 'Kiwi', 'Orange']
```

Lists

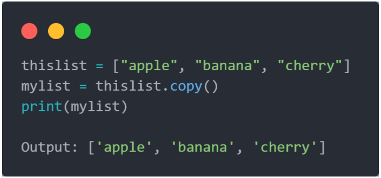
- Reverse Order
  - What if you want to reverse the order of a list, regardless of the alphabet?
  - The **reverse()** method reverses the current sorting order of the elements.



```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)

Output: ['cherry', 'Kiwi', 'Orange', 'banana']
```

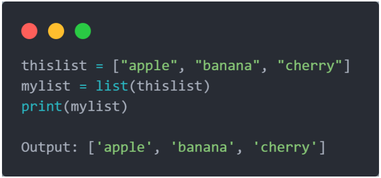
Lists

- Copy a List
  - You cannot copy a list simply by typing **list2 = list1**, because: **list2** will only be a reference to **list1**, and changes made in **list1** will automatically also be made in **list2**.
  - There are ways to make a copy, one way is to use the built-in List method **copy()**.



```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)

Output: ['apple', 'banana', 'cherry']
```

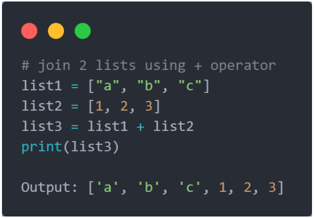
  - Another way to make a copy is to use the built-in method **list()**.



```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)

Output: ['apple', 'banana', 'cherry']
```

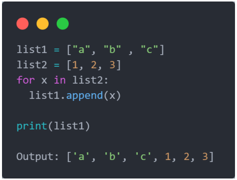
Lists

- Join Two Lists
  - There are several ways to join, or concatenate, two or more lists in Python.
  - One of the easiest ways are by using the + operator.



```
# join 2 lists using + operator
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)

Output: ['a', 'b', 'c', 1, 2, 3]
```

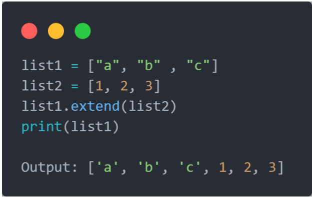
  - Another way to join two lists is by appending all the items from list2 into list1, one by one.



```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
  list1.append(x)

print(list1)

Output: ['a', 'b', 'c', 1, 2, 3]
```

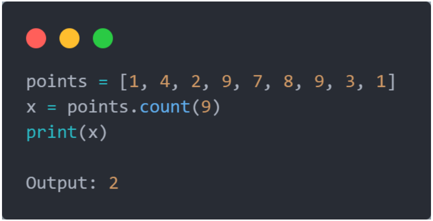
Lists

- Or you can use the **extend()** method, where the purpose is to add elements from one list to another list:



```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)

Output: ['a', 'b', 'c', 1, 2, 3]
```

- You can count the number of a specified value in a list by using **count()** function.



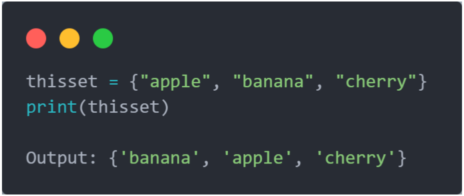
```
points = [1, 4, 2, 9, 7, 8, 9, 3, 1]
x = points.count(9)
print(x)

Output: 2
```

Table of Contents

1. Lambda Functions
2. Arrays
3. List
4. Sets
5. Tuples
6. Dictionaries
7. Short Summary

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
- * **Note:** Set items are unchangeable, but you can remove items and add new items.
- Sets are written with curly brackets.

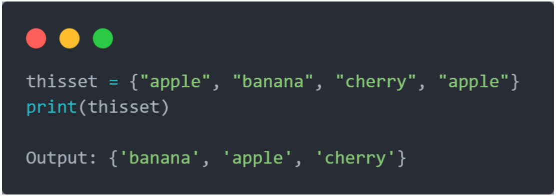


```
thisset = {"apple", "banana", "cherry"}
print(thisset)

Output: {'banana', 'apple', 'cherry'}
```

- Set items are unordered, unchangeable, and do not allow duplicate values.
- Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

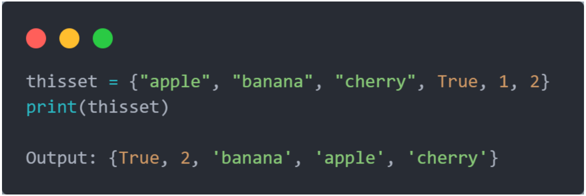
Sets

- Sets cannot have two items with the same value.



```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)

Output: {'banana', 'apple', 'cherry'}
```

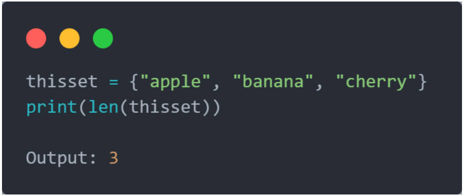
- **Note:** The values True and 1 are considered the same value in sets, and are treated as duplicates



```
thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)

Output: {True, 2, 'banana', 'apple', 'cherry'}
```

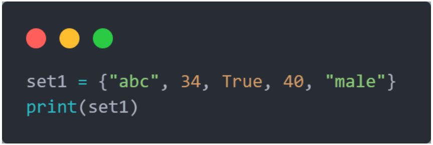
Sets

- To determine how many items a set has, use the **len()** function.



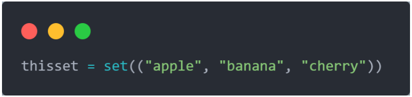
```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))

Output: 3
```

- Set items can be of any data type.
- A set can contain different data types.



```
set1 = {"abc", 34, True, 40, "male"}
print(set1)
```

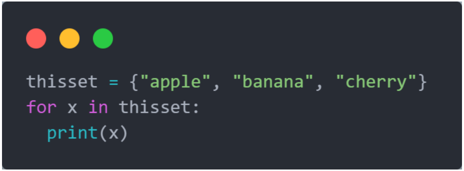
- It is also possible to use the set() constructor to make a set.



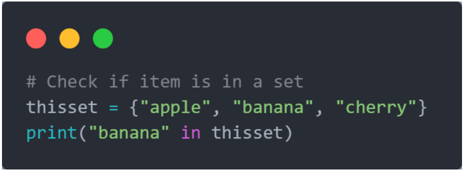
```
thisset = set(("apple", "banana", "cherry"))
```

Sets

- Access Items:
  - You cannot access items in a set by referring to an index or a key.
  - But you can loop through the set items using a **for** loop, or ask if a specified value is present in a set, by using the **in** keyword.



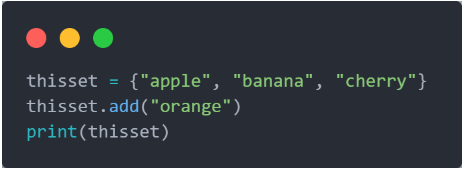
```python
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```



```python
# Check if item is in a set
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

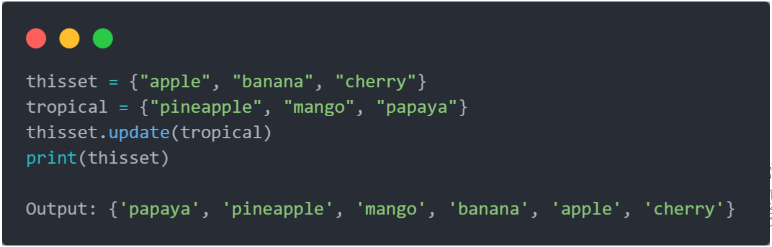
Sets

- Add Items:
  - To add one item to a set use the **add()** method.



```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

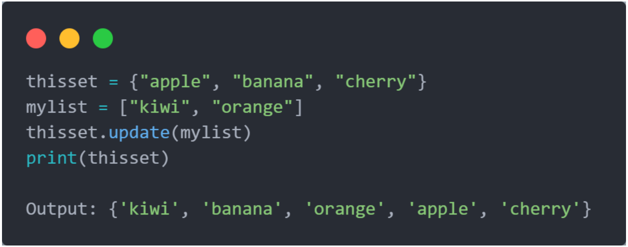
- Add Sets:
  - To add items from another set into the current set, use the **update()** method.



```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)

Output: {'papaya', 'pineapple', 'mango', 'banana', 'apple', 'cherry'}
```

Sets

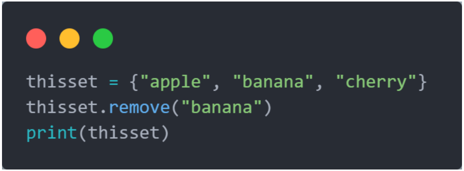
- Add Any Iterable:
    - The object in the **update()** method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).



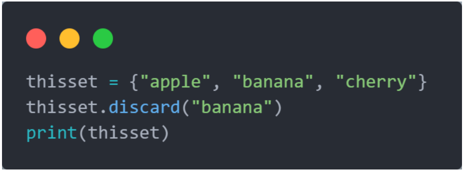
```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)

Output: {'kiwi', 'banana', 'orange', 'apple', 'cherry'}
```

Sets

- Remove Item
  - To remove an item in a set, use the **remove()**, or the **discard()** method.



```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

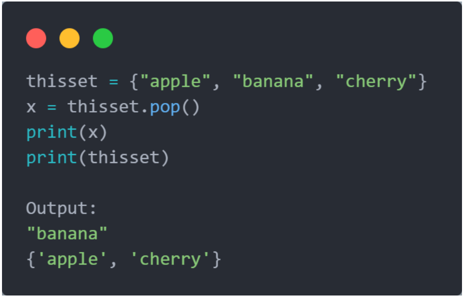


```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

  - **Note:** If the item to remove does not exist, **remove()** will raise an error but **discard()** will **NOT** raise an error.

Sets

- You can also use the **pop()** method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.
- The return value of the **pop()** method is the removed item.

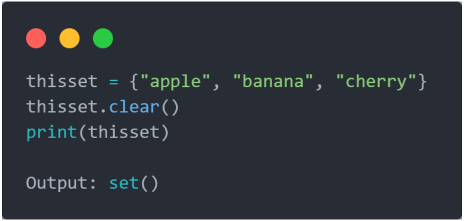


```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

Output:
"banana"
{'apple', 'cherry'}
```

- **Note:** Sets are unordered, so when using the pop() method, you do not know which item that gets removed.

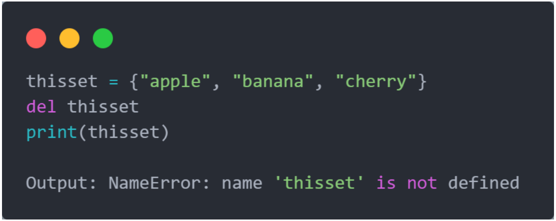
Sets

- The **clear()** method empties the set



```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)

Output: set()
```

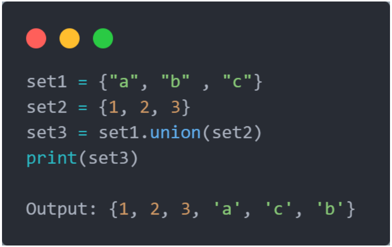
- The **del** keyword will delete the set completely



```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)

Output: NameError: name 'thisset' is not defined
```

Sets

- Join Two Sets
  - There are several ways to join two or more sets in Python.
  - You can use the **union()** method that returns a new set containing all items from both sets, or the **update()** method that inserts all the items from one set into another

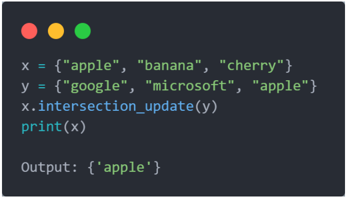


```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)

Output: {1, 2, 3, 'a', 'c', 'b'}
```

  - The update() method inserts the items in set2 into set1.
  - **Note:** Both union() and update() will exclude any duplicate items.

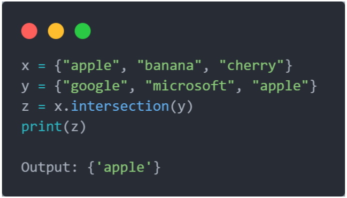
Sets

- Keep ONLY the Duplicates
  - The **intersection_update()** method will keep only the items that are present in both sets.



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)

Output: {'apple'}
```

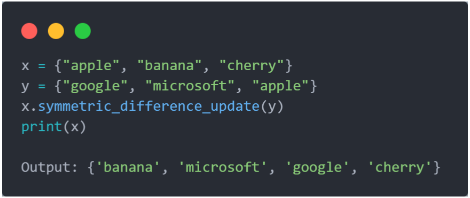
  - The **intersection()** method will return a new set, that only contains the items that are present in both sets.



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)

Output: {'apple'}
```

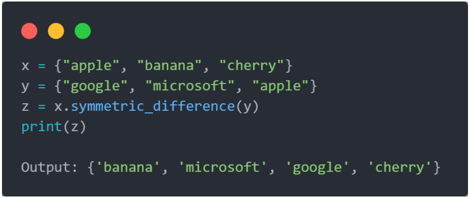
Sets

- Keep All, But NOT the Duplicates:
  - The **symmetric_difference_update()** method will keep only the elements that are NOT present in both sets.



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)

Output: {'banana', 'microsoft', 'google', 'cherry'}
```

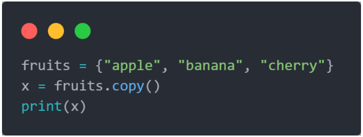
  - The **symmetric_difference()** method will return a new set, that contains only the elements that are NOT present in both sets.



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)

Output: {'banana', 'microsoft', 'google', 'cherry'}
```

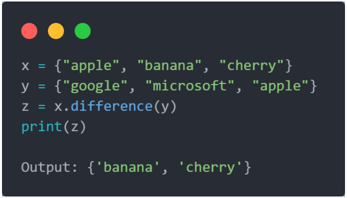
Sets

- Python Set **copy()** Method returns a copy of the set



```
fruits = {"apple", "banana", "cherry"}
x = fruits.copy()
print(x)
```

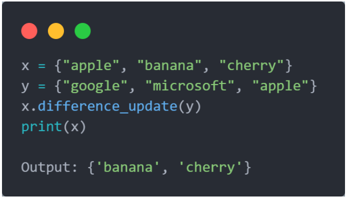
- Python Set **difference()** Method returns a set containing the difference between two or more sets



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z)

Output: {'banana', 'cherry'}
```

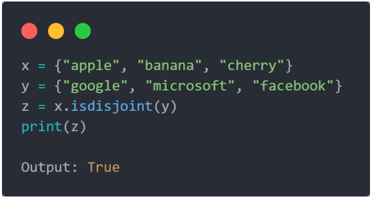
Sets

- Python Set **difference_update()** Method removes the items in this set that are also included in another, specified set



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.difference_update(y)
print(x)

Output: {'banana', 'cherry'}
```

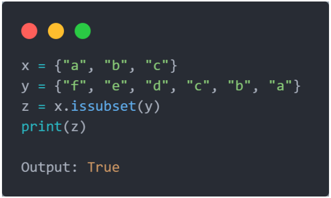
- Python Set **isdisjoint()** Method returns whether two sets have a intersection or not



```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "facebook"}
z = x.isdisjoint(y)
print(z)

Output: True
```

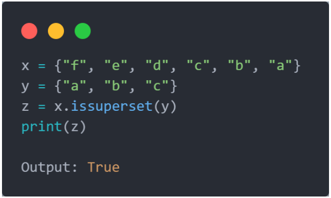
Sets

- Python Set **issubset()** Method returns whether another set contains this set or not



```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}
z = x.issubset(y)
print(z)

Output: True
```

- Python Set issuperset() Method returns whether this set contains another set or not



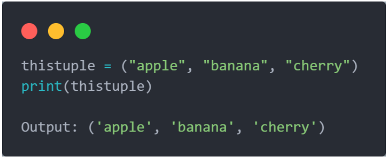
```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)
print(z)

Output: True
```

Table of Contents

1. Lambda Functions
2. Arrays
3. List
4. Sets
5. Tuples
6. Dictionaries
7. Short Summary

Tuples

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is **ordered** and **unchangeable**.
- Tuples are written with round brackets.

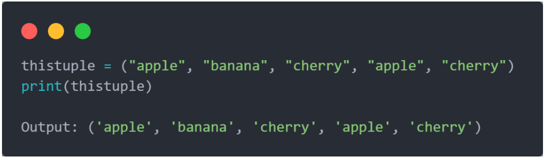


```
thistuple = ("apple", "banana", "cherry")
print(thistuple)

Output: ('apple', 'banana', 'cherry')
```

- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuple items:
  - Tuple items are ordered, unchangeable, and allow duplicate values.
  - Tuple items are indexed, the first item has index **[0]**, the second item has index **[1]** etc.

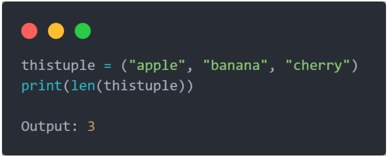
Tuples

- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Since tuples are indexed, they can have items with the same value:



```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)

Output: ('apple', 'banana', 'cherry', 'apple', 'cherry')
```

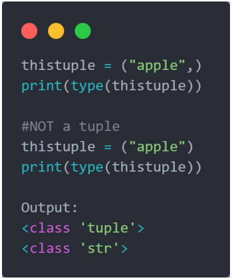
- To determine how many items a tuple has, use the **len()** function:



```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))

Output: 3
```

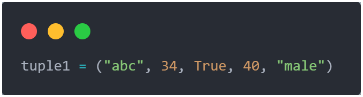
Tuples

- Create tuple with one item
  - To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.



```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))

Output:
<class 'tuple'>
<class 'str'>
```

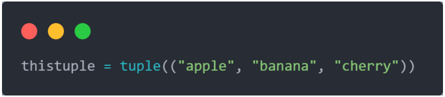
- Tuple items can be of any data type
- A tuple can contain different data types



```
tuple1 = ("abc", 34, True, 40, "male")
```

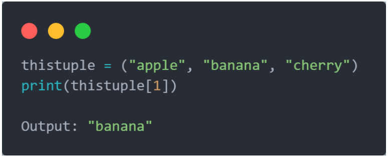
Tuples

- It is also possible to use the **tuple()** constructor to make a tuple.

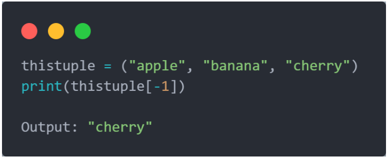


```
thistuple = tuple(("apple", "banana", "cherry"))
```

- You can access tuple items by referring to the index number, inside square brackets. Negative indexing means start from the end.



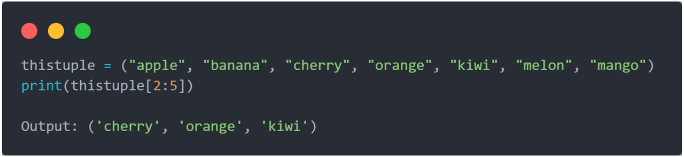
```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])

Output: "banana"
```



```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])

Output: "cherry"
```

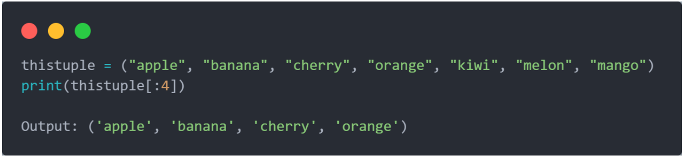
Tuples

- Range of Indexes:
  - You can specify a range of indexes by specifying where to start and where to end the range. You can specify a range of indexes by specifying where to start and where to end the range.



```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])

Output: ('cherry', 'orange', 'kiwi')
```

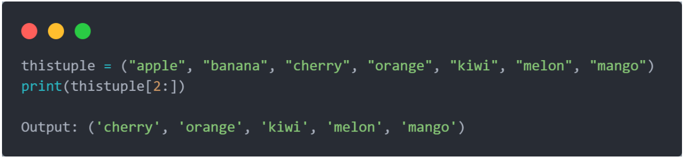
  - By leaving out the start value, the range will start at the first item:



```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])

Output: ('apple', 'banana', 'cherry', 'orange')
```

Tuples

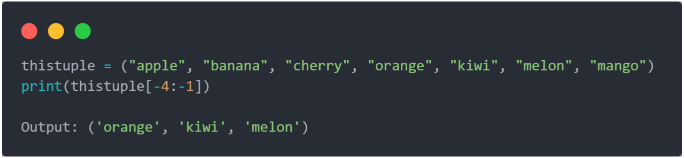
- By leaving out the end value, the range will go on to the end of the list:



```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])

Output: ('cherry', 'orange', 'kiwi', 'melon', 'mango')
```

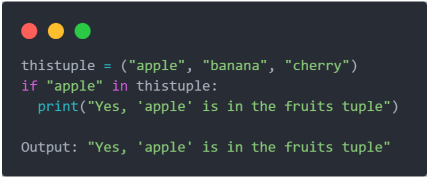
Tuples

- Range of Negative Indexes:
  - Specify negative indexes if you want to start the search from the end of the tuple:



```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])

Output: ('orange', 'kiwi', 'melon')
```

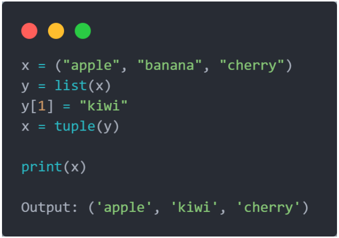
- To determine if a specified item is present in a tuple use the **in** keyword:



```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")

Output: "Yes, 'apple' is in the fruits tuple"
```

Tuples

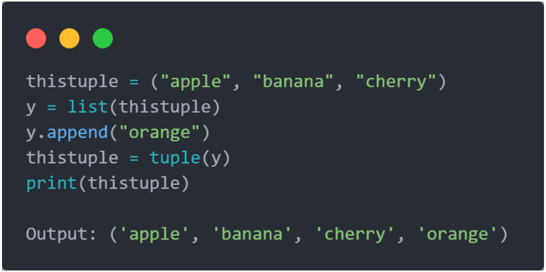
- Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created. But there are some workarounds.
- You can convert the tuple into a list, change the list, and convert the list back into a tuple.



```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)

Output: ('apple', 'kiwi', 'cherry')
```

Tuples

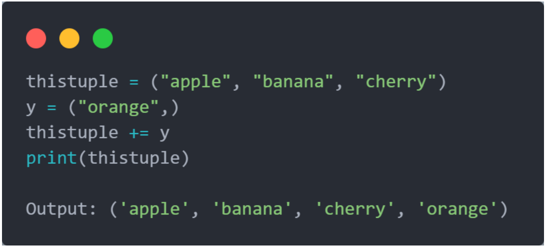
- Add items to tuples: Since tuples are immutable, they do not have a built-in **append()** method, but there are other ways to add items to a tuple:
    - **Convert into a list**: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.



```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
print(thistuple)

Output: ('apple', 'banana', 'cherry', 'orange')
```

Tuples

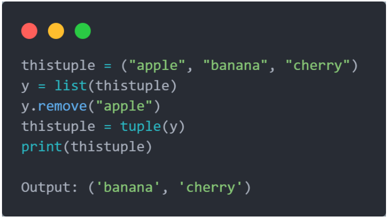
- **Add tuple to a tuple**. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:



```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)

Output: ('apple', 'banana', 'cherry', 'orange')
```

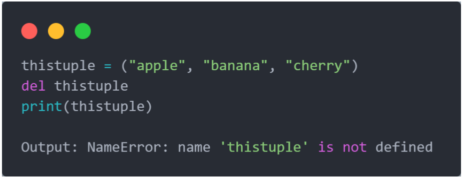
Tuples

- Remove items:
  - Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items by convert to a list for removing and convert back to the tuple type:



```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
print(thistuple)

Output: ('banana', 'cherry')
```

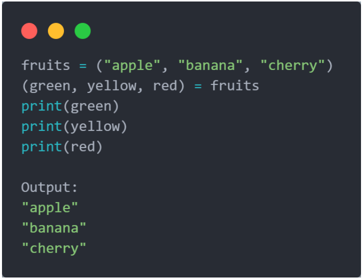
  - Or you can delete the tuple completely using **del** keyword:



```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple)

Output: NameError: name 'thistuple' is not defined
```

Tuples

- Unpacking a Tuple:
  - When we create a tuple, we normally assign values to it. This is called ”packing” a tuple
  - But, in Python, we are also allowed to extract the values back into variables. This is called ”unpacking”:

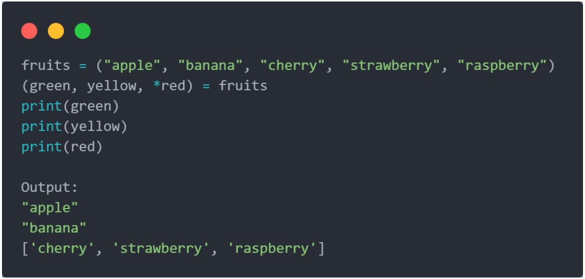


```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)

Output:
"apple"
"banana"
"cherry"
```

  - **Note:** The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Tuples

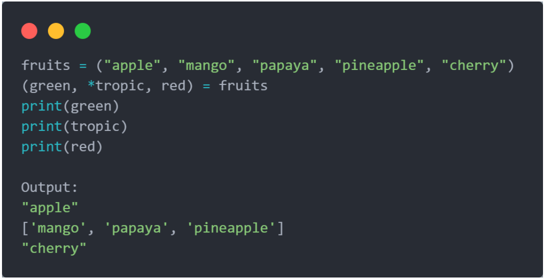
- Using Asterisk *
  - If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list



```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)

Output:
"apple"
"banana"
['cherry', 'strawberry', 'raspberry']
```

Tuples

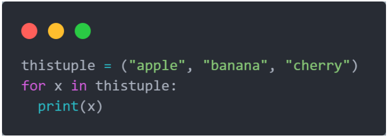
- If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.



```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)

Output:
"apple"
['mango', 'papaya', 'pineapple']
"cherry"
```

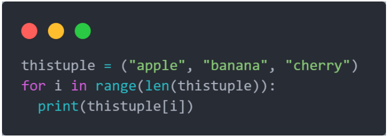
Tuples

- Loop Through a Tuple
  - You can loop through the tuple items by using a **for** loop.



```python
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

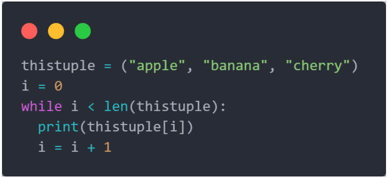
- Loop Through the Index Numbers
  - You can also loop through the tuple items by referring to their index number.
  - Use the **range()** and **len()** functions to create a suitable iterable.



```python
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Tuples

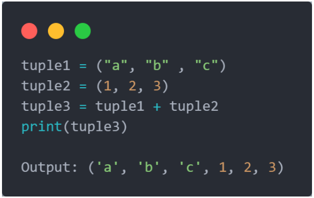
- Using a While Loop
  - You can loop through the tuple items by using a **while** loop.
  - Use the **len()** function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.
  - Remember to increase the index by 1 after each iteration.



```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

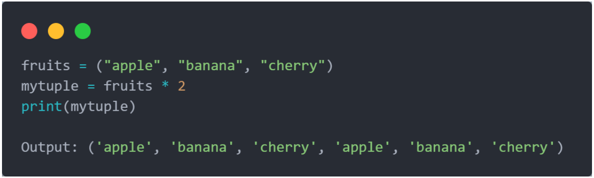
Tuples

- Join Two Tuples:
  - To join two or more tuples you can use the + operator:



```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)

Output: ('a', 'b', 'c', 1, 2, 3)
```

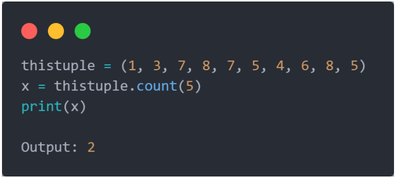
- Multiply Tuples
  - If you want to multiply the content of a tuple a given number of times, you can use the * operator:



```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)

Output: ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

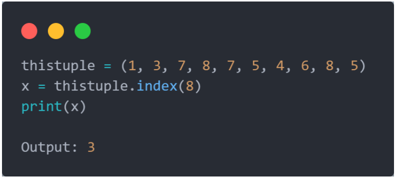
Tuples

- Python Tuple count() Method returns the number of times a specified value occurs in a tuple



```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)

Output: 2
```

- Python Tuple index() Method searches the tuple for a specified value and returns the position of where it was found



```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)
print(x)

Output: 3
```

The End

THE END!!!